NumLin: Linear Types for Linear Algebra

Dhruv C. Makwana[1 [orcidID]] and Neelakantan R. Krishnaswami[2 [orcidID]]

[1] dcm41@cam.ac.uk dhruvmakwana.com
[2] Department of Computer Science, University of Cambridge
nk480@cl.cam.ac.uk

Abstract. Briefly summarize the contents of the paper in 150–250 words.

Keywords: numerical, linear, algebra, types, permissions, OCaml

1 Introduction

NumLin is a functional programming language designed to express the APIs of low-level linear algebra libraries (such as BLAS/LAPACK) safely and explicitly. It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs.

NumLin allows a novice to understand and work with complicated linear algebra library APIs, as well as point out subtle aliasing bugs and reduce memory usage in existing programs. In fact, we were able to use NumLin to find linearity and aliasing bugs in a linear algebra program that was *generated* by another program *specifically designed to translate matrix expressions into an efficient sequence of calls to linear algebra routines*. We were also able to reduce the number of temporaries used by the same program, using NumLin's type system to guide us.

NumLin's implementation supports several syntactic conveniences as well as a *usable* integration with real OCaml libraries.

1.1 Contributions

In this paper

- we describe NumLin, a linearly typed language for linear algebra programs
- we illustrate that NumLin's design and features are well-suited to its intended domain with progressively sophisticated examples
- we prove NumLin's soundness, using a step-indexed logical relation
- we describe a very simple, unification based type-inference algorithm for polymorphic fractional permissions (similar to ones used for parametric polymorphism), demonstrating an alternative approach to dataflow analysis [1]
- we describe an implementation that is both compatible with and usable from existing code

- we show an example of how using NumLin helped highlight linearity and aliasing bugs, and reduce the memory usage of a *generated* linear algebra program
- we show that using NumLin, we can achieve parity with C for linear algebra routines, whilst having much better static guarantees about the linearity and aliasing behaviour of our programs.

2 NumLin Overview and Examples

2.1 Overview

Linearity is at the heart of NumLin. Linearity allows us to express a pure-functional API for numerical library routines that mutate arrays and matrices. Linearity also restricts aliasing of (values which represent) pointers.

Intuitionism: ! and `Many` However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them *intuitionistically*, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding `Many` constructor and `let Many <id> = .. in ..` eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NumLin enforces simple syntactic restrictions on which values can be wrapped up in a `Many` constructor (details in Section 3).

Fractional Permissions There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine `gemm`, which (rather tersely) stands for *GEneric Matrix Multiplication*. A *simplified* definition of `gemm(`α`, A, B,` β`, C)` is $C := \alpha AB + \beta C$. In this case, `A` and `B` may alias each other but neither may alias `C`, because it is being written to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a freed value is *out of scope* for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing *without* losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are *parameterised* by a *fraction*. A fraction is either 1 (2^0) or exactly *half* of another fraction (2^{-k}, for natural k). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a *borrow*: the programmer may read from the value but not write to or free it. Creating an

array/matrix gives you ownership of it, so too does having one (with a fractional permission of 2^0) passed in as an argument.

In NumLin, we can produce two aliases of a single array/matrix, by *sharing* it. If the original alias had a fractional permission of 2^{-k} then the two new aliases of it will have a fractional permission of $2^{-(k+1)}$ each. Thanks to linearity, the original array/matrix with a fractional permission of 2^{-k} will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of `free` and `set` (for mutation) require a *whole* (2^0) permission.

If we have two aliases *to the same matrix* with *identical* fractional permissions ($2^{-(k+1)}$), we can recombine or *unshare* them back into a single one, with a larger 2^{-k} permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

Runtime Errors Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that can fail at runtime (the other being dimension checks, such as for `gemm`). The check being performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

Recursion The final feature of NumLin which makes it sufficiently expressive is recursion (and of course, conditional branches to ensure termination). Conditional branches are implemented by ensuring that both branches use the same set of linear values. A function can be recursive if it captures no linear values from its environment. Like with `Many`, this is enforced via simple syntactic restrictions on the definition of recursive functions.

2.2 Examples

Factorial Although a factorial function (Figure 1) may seem like an aggressively pedestrian first example, in a linearly typed language such as NumLin it represents the culmination of many features.

To simplifiy the design and implementation of NumLin's type system, recursive functions must have full type annotations (non-recursive functions need only their argument types annotated). Its body is a closed expression (with respect to the function's arguments), so it type-checks (since it does not capture any linear values from its environment).

The only argument is `!x : !int`. The ! annotation on `x` is a syntactic convenience for declaring the value to used intuitionistically, its full and precise meaning is described in Section 4.1.

The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it does not). Any linear values used by the condition would not be in scope in either branch of the `if`-expression. Both branches use `x` differently: one ignores it completely and the other uses it twice.

```
let rec factorial ( !x : !int ) : !int =
    if x < 0 || x = 0 then
        1
    else
        x * factorial (x - 1) in factorial
;;
```

Fig. 1. Factorial function in NumLin.

```
let rec sum_array (!i : !int) (!n : !int) (!x0 : !elt)
                  ('x) (row : 'x arr) : 'x arr * !elt =
    if i = n then
        (row, x0)
    else
        let (row, !x1) = row[i] in
        sum_array (i + 1) n (x0 +. x1) 'x row in
    sum_array
;;
```

Fig. 2. Summing over an array in NumLin.

All numeric and boolean literals are implicitly wrapped in a `Many` and all primitives involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuting `||` behaves in exactly the same way as a boolean-valued `if`-expression.

Summing over an Array Now we can add fractional permissions to the mix: Figure 2 shows a simple, tail-recursive implementation of summing all the elements in an array. There are many new features; first among them is `!x0 : !elt`, the type of array/matrix elements (64-bit floating point).

Second is `('x) (row: 'x arr)` which is an array with a universally-quantified fractional permission. In particular, this means the body of the function cannot mutate or free the input array, only read from it. If the programmer did try to mutate or free `row`, then they would get a helpful error message (Figure 3).

Alongside taking a `row: 'x arr`, the function also returns an array with exactly the same fractional permission as the `row` (which can only be `row`). This is necessary because of linearity: for the caller, the original array passed in as an argument would be out of scope for the rest of the expression, so it needs to be returned and then rebound to be used for the rest of the function.

An example of this consuming and re-binding is in `let (row, !x1) = row[i]`. Indexing is implemented as a primitive `get: 'x. 'x arr --o !int --o 'x arr * !elt`. Although fractional permissions can be passed around explicitly (as done in the recursive call), they can also be *automatically inferred at call sites*: `row[i] == get _ row i` takes advantage of this convenience.

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                          *)
(*     z arr                                                         *)
(* with                                                              *)
(*     'x arr                                                        *)
(*                                                                   *)
(* Var 'x is universally quantified                                  *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In test/examples/sum_array.lt, at line: 7 and column: 19        *)
```

Fig. 3. Attempting to write to or free a read only array in NumLin.

```
let rec simp_oned_conv
        (!i : !int) (!n : !int) (!x0 : !elt)
        (write : z arr) ('x) (weights : 'x arr)
        : 'x arr * z arr =
    if n = i then (weights, write) else
    let !w0 <- weights[0]   in
    let !w1 <- weights[1]   in
    let !w2 <- weights[2]   in
    let !x1 <- write[i]     in
    let !x2 <- write[i + 1] in
    let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
    simp_oned_conv (i + 1) n x1 written _ weights in
  simp_oned_conv
;;
```

Fig. 4. *Simplified* one-dimensional convolution.

One-dimensional Convolution Figure 4 extends the set of features demonstrated by the previous examples by mutating one of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel (array of weights) and an input vector. It modifies the input vector *in-place* by replacing each `write[i]` with a weighted (as per the values in the kernel) sum of it and its neighbours; intuitively, sliding a dot-product with the kernel across the vector.

What's implemented in Figure 4 is a *simplified* version of this idea, so as to not distract from the features of NumLin. The simplifications are:

- the kernel has a length 3, so only the value of `write[i-1]` (prior to modification in the previous iteration) needs to be carried forward using `x0`
- `write` is assumed to have length `n+1`
- `i`'s initial value is assumed to be `1`
- `x0`'s initial value is assumed to be `write[0]`
- the first and last values of `write` are ignored.

Mutating an array is implemented similarly to indexing one: a primitive `set : z arr --o !int --o !elt --o z arr`. It consumes the original array and returns a new array with the updated value. `let written = write[i] := <exp>` is just syntactic sugar for `let written = set write i <exp>`.

```
let !square ('x) (x : 'x mat) =
    let (x, (!m, !n)) = sizeM _ x in
    let (x1, x2) = shareM _ x in
    let answer <- new (m, n) [| x1 * x2 |] in
    let x = unshareM _ x1 x2 in
    (x, answer) in
    square
;;
```

Fig. 5. Linear regression (OLS): $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

Since `write: z arr` (where `z` stands for $k = 0$, representing a fractional permission of $2^{-k} = 2^{-0} = 1$), we may mutate it, but since we only need to read from `weights`, its fractional permission index can be universally-quantified. In the recursive call, we see _ being used explicitly to tell the compiler to *infer* the correct fractional permission based on the given arguments.

Squaring a Matrix *The most pertinent aspect of* NumLin *is the types of its primitives.* While the types of operations such as `get` and `set` might be borderline obvious, the types of BLAS/LAPACK routines become an *incredibly useful, automated check for using the API correctly.*

Figure 5 shows how a linearly-typed matrix squaring function may be written in NumLin. It is a *non-recursive* function declaration (the return type is inferred). Since we would like to be able to use a function like `square` more than once, it is marked with a `!` annotation (which also ensures it captures no linear values from the surrounding environment).

To square a matrix, first, we extract the dimensions of the argument `x`. Then, because we need to use `x` twice (so that we can multiply it by itself) but linearity only allows one use, we use `shareM: 'x. 'x mat -o 'x s mat * 'x s mat` to split the permission `'x` (which represents 2^{-x}) into two halves (`'x s`, which represents $2^{-(x+1)}$).

Even if `x` had type `z mat`, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NumLin, is always of type `z mat`) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 3.

The line `let answer <- new (m,n) [| x1 * x2 |]` is syntactic sugar for first creating a new $m \times n$ matrix (`let answer = matrix m n`) and then storing the result of the multiplication in it (`let ((x1, x2), answer) = gemm 1. _ (x1, false) _ (x2, false) 0. answer`). `false` means the matrix should not be accessed with indices transposed.

By using some simple pattern-matching and syntactic sugar, we can:

- write normal-looking, apparently non-linear code
- use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK routine inserted with appropriate re-bindings

```
let !lin_reg ('x) (x : 'x mat)
             ('y) (y : 'y mat) =
  let (x, (!_n, !m)) = sizeM _ x in
  let xy <- new (m, 1) [| x^T * y |] in
  let x_T_x <- new (m, m) [| x^T * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg
;;
```

Fig. 6. Linear regression (OLS): $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

– retain the safety of linear types with fractional permissions by having the compiler statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK routines.

Linear Regression In Figure 6, we wish to compute $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$. To do that, first, we extract the dimensions of matrix `x`. Then, we say we would like `xy` to be a new matrix, of dimension $m \times 1$, which contains the result of $\mathbf{X^T y}$ (using syntactic sugar for `matrix` and `gemm` calls similar to that used in Figure 5, with a `^T` annotation on `x` to set `x`'s 'transpose indices'-flag to `true`).

However, the line `let x_T_x <- new (m,m) [| x^T * x |]`, works for a slightly different reason: that pattern is matched to a BLAS call to (`syrk true 1. x 0. x_T_x`), which only uses `x` once. Hence `x` can appear *twice* in the *pattern* without any calls to `share`.

After computing `x_T_x`, we need to invert it and then multiply it by `xy`. The BLAS routine `posv: z mat --o z mat --o z mat * z mat` does exactly that: assuming the first argument is symmetric, `posv` mutates its second argument to contain the desired value. Its first argument is also mutated to contain the (upper triangular) Cholesky decomposition factor of the original matrix. Since we do not need that matrix (or its memory) again, we `free` it. If we forgot to, we would get a `Variable to_del not used` error. Lastly, we return the `answer` alongside the untouched input matrices `(x,y)`.

L1-Norm Minimisation on Manifolds L1-Norm minimisation is often used in optimisation problems, as a *regularisation* term for reducing the influence of outliers. Although the below formuation[2] is intended to be used with *sparse* computations, NumLin's current implementation only implements dense ones. However, it still serves as a useful example of explaining NumLin's features.

Figure 7 shows even more pattern-matching. Patterns of the form `let <id> <- [| beta * c + alpha * a * b |]` are also desugared to `gemm` calls. Primitives like `transpose: 'x. 'x mat --o 'x mat * z mat` and `eye: !int --o z mat` allocate new matrices; `transpose` returns the transpose of a given matrix and `eye k` evaluates to a $k \times k$ identity matrix.

```
let !l1_norm_min (q : z mat) (u : z mat) =
  let (u, (!_n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n , q_inv_u ) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T ) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min
;;
```

Fig. 7. L1-norm minimisation on manifolds: $\mathbf{Q^{-1}U(I + U^T Q^{-1} U)^{-1} U^T}$

We also see our first example of re-using memory for different matrices: like with `to_del` and `posv` in the previous example, we do not need the value stored in `tmp_5_5` after the call to `gesv` (a primitive similar to `posv` but for a non-symmetric first argument). However, we can re-use its memory much later to store `answer` with `let answer <- [| 0. * tmp_5_5 + q_inv_u * inv_u_T |]`. Again, thanks to linearity, the identifiers `q` and `tmp_5_5` are out of scope by the time `answer` is bound. Although during execution, all three refer to the same piece of memory, logically they represent different values throughout the computation.

Kalman Filter A *Kalman Filter*[3] is a an algorithm for combining prior knowledge of a state, a statistical model and measurements from (noisy) sensors to produce an estimate a more reliable estimated of the current state. It has various applications (navigation, signal-processing, econometrics) and is relevant here because it is usually presented as a series of complex matrix equations.

Figure 8 shows a NumLin implementation of a Kalman filter (equations in Figure 9). A few new features and techniques are used in this implementation:

- `sym` annotations in matrix expressions: when this is used, a call to `symm` (the equivalent of `gemm` but for symmetric matrices so that only half the operations are performed) is inserted
- `copyM_to` is used to re-use memory by *overwriting* the contents of its second argument to that of its first (erroring if dimensions do not match)
- `let new_r <- new [| r_2 |]` creates a copy of `r_2`
- `potrs _chol_r data_2` uses a pre-computed Cholesky decomposition to multiply `data_2` $= H\mu -$ data by `r_2` $= (R + H\Sigma H^T)^{-1}$
- a lot of memory re-use; the following sets of identifiers alias each other:
  - `r_1` and `r_2`
  - `data_1`, `data_2` and `sol_data`

```
      let !kalman
        ('s) (sigma : 's mat) (* n,n *)
        ('h) (h : 'h mat)     (* k,n *)
        ('m) (mu : 'm mat)    (* n,1 *)
        (r_1 : z mat)         (* k,k *)
        (data_1 : z mat)      (* k,1 *) =
        let (h, (!k, !n)) = sizeM _ h in
(*16*)  let sigma_h <- new (k, n) [| h * sym (sigma) |] in
(*17*)  let r_2 <- [| sigma_h * h^T + r_1 |] in
(*18*)  let data_2 <- [| h * mu - data_1 |] in
(*19*)  let (h, new_h) = copyM_to _ h sigma_h in
(*20*)  let new_r <- new [| r_2 |] in
(*21*)  let (chol_r, sol_h) = posv new_r new_h in
(*23*)  let (chol_r, sol_data) = potrs _ chol_r data_2 in
        let () = freeM (* k,k *) chol_r in
(*24*)  let h_sol_h <- new (n, n) [| h^T * sol_h |] in
        let () = freeM (* k,n *) sol_h in
(*25*)  let h_sol_data <- new (n, 1) [| h^T * sol_data |] in
(*26*)  let mu_copy <- new [| mu |] in
(*27*)  let new_mu <- [| sym (sigma) * h_sol_data + mu_copy |] in
        let () = freeM (* n,1 *) h_sol_data in
(*28*)  let h_sol_h_sigma <- new (n,n) [| h_sol_h * sym(sigma) |] in
(*29*)  let (sigma, sigma_copy) = copyM_to _ sigma h_sol_h in
(*30*)  let new_sigma <- [| sigma_copy - sym (sigma) * h_sol_h_sigma |] in
        let () = freeM (* n,n *) h_sol_h_sigma in
        ((sigma, (h, (mu, (r_2, sol_data)))), (new_mu, new_sigma)) in
      kalman
      ;;
```

Fig. 8. Kalman filter: see Figure 9 for the equations this code implements. Line numbers in comments refer to equivalent lines in a C implementation (Figure 18).

- `new_h` and `sol_h`
- `h_sol_h`, `sigma_copy` and `new_sigma`
- `mu_copy` and `new_mu`.

The NumLin implementation is much longer than the mathematical equations for two reasons. First, the NumLin implementation is a let-normalised form of the Kalman equations: since there a large number of unary/binary (and occasionally ternary) sub-expressions in the equations, naming each one line at a time makes the implementation much longer. Second, NumLin has the additional task of handling explicit allocations, aliasing and frees of matrices. However, it is exactly this which makes it possible (and often, easy) to spot additional opportunities for memory re-use. Furthermore, a programmer can explore those opportunities easily because NumLin's type system statically enforces correct memory management and the aliasing assumptions of BLAS/LAPACK routines.

$$\mu' = \mu + \Sigma H^T (R + H\Sigma H^T)^{-1} (H\mu - \text{data})$$
$$\Sigma' = \Sigma (I - H^T (R + H\Sigma H^T)^{-1} H\Sigma)$$

Fig. 9. Kalman filter equations (credit: matthewrocklin.com).

3 Formal System

3.1 Core Type Theory

The full typing rules are in Appendix , but the key ideas are as follow: Perm, Type, Θ, Δ, Γ, value restriction, lambda, Gen/Spc, and Fix.

3.2 Dynamic Semantics

and operational semantics here

3.3 The logical relation

Describe the step-indexed logical relation and its main properties

3.4 Soundness Theorem

State the fundamental lemma, and sketch the proof a little

4 Implementation

Talk about how you implemented NumLin and the general architecture. Talk about how simple everything is, and also about how implementing inference for fractions is.

4.1 Implementation Strategy

NumLin transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although NumLin's current implementation is not as an embedded DSL, its the general design is simple enough to adapt to being so and also to target other languages.

Alongside the transpiler, a 'Read-Check-Translate' loop, benchmarking program and a test suite are included in the artifacts accompanying this paper.

1. **Parsing**. A generated, LR(1) parser parses a text file into a syntax tree. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support.

2. **Desugaring**. The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.
3. **Matrix Expressions** are also desugared into the abstract syntax tree through some simple pattern-matching.
4. **Type checking**. The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.
5. **Code Generation**. The abstract syntax tree is translated into OCaml, with a few 'optimisations' to produce more readable code. This process is type-preserving: NumLin's type system is embedded into OCaml's (Figure 11), and so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use NumLin is to have the build system generate code at *compile-time* and then have the generated code be used by other modules like normal OCaml functions. This makes it possible and even easy to use NumLin alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in NumLin.

Desugaring, Matrix Expressions and Type Checking Desugaring is conventional, outlined in Figure 16. Matrix expression are translated into BLAS/LAPACK calls via purely syntactic pattern-matching, outlined in Figure 10.

Type checking is mostly standard for a linearly typed language, with the exception of fractional permission inference. Because all functions must have their argument types explicitly annotated, inferring the correct fraction at a call-site is simply a matter of unification.

There are a few differences between the type system as presented in 3.2 and how we implemented it: the environment *changes* as a result of type checking an expression (the standard transformation to avoid a non-deterministic split of the environment for checking pairs); variables are *marked as used* rather than removed for better error messages; variables are *tagged* as linear or intuitionistic in *one* environment as opposed to being stored in *two* separate ones (this allows scoping/variable look-up to be handled uniformly).

What more can/should we say here?

Code Generation is a straightforward mapping from NumLin's core constructs to high-level OCaml ones. We embed NumLin's type- and term- constructors into OCaml as a sanity check on the output (Figure 11).

This is also useful when using NumLin from within OCaml; for example, we can use existing tools to inspect the type of the function we are using (Figure 12). It is worth reiterating that only the type- and term- constructors are translated into OCaml, NumLin's precise control over linearity and aliasing are not brought over.

We actually use this fact to our advantage to clean up the output OCaml by removing what would otherwise be redundant re-bindings (Figure 13). Combined

$$\textbf{let } v \leftarrow x[e] \textbf{ in } e \Rightarrow \textbf{let } (x, !v) \ = x[e] \textbf{ in } e \qquad \text{(similarly for matrices)}$$
$$\textbf{let } x_2 \leftarrow \textbf{new } [|\ x_1\ |] \textbf{ in } e \Rightarrow \textbf{let } (x_1, x_2) \ = \textbf{copyM } \_\ x_1 \textbf{ in } e$$
$$\textbf{let } x_2 \leftarrow [|\ x_1\ |] \textbf{ in } e \Rightarrow \textbf{let } (x_1, x_2) \ = \textbf{copyM\_to } \_\ x_1\ x_2 \textbf{ in } e$$

$$M ::= X \ \mid \ X^T \ \mid \ \mathrm{sym}(X)$$

$$\textbf{let } Y \leftarrow \textbf{new } (n, k)\ [|\ \alpha M_1 M_2\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } Y \ = \textbf{matrix } n\ k \textbf{ in let } Y \leftarrow [|\ \alpha M_1 M_2 + 0Y\ |] \textbf{ in } e$$
$$\textbf{let } Y \leftarrow [|\ \alpha X X^T + \beta Y\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } (X, Y) \ = \textbf{syrk false } \alpha\ \_\ X\ \beta\ Y \textbf{ in } e$$
$$\textbf{let } Y \leftarrow [|\ \alpha X^T X + \beta Y\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } (X, Y) \ = \textbf{syrk true } \alpha\ \_\ X\ \beta\ Y \textbf{ in } e$$
$$\textbf{let } Y \leftarrow \ [|\ \alpha\, \mathrm{sym}(X_1)\, X_2 + \beta Y\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } ((X_1, X_2), Y) \ = \textbf{symm false } \alpha\ \_\ X_1 \_\ X_2\ \beta\ Y \textbf{ in } e$$
$$\textbf{let } Y \leftarrow \ [|\ \alpha\, X_2\, \mathrm{sym}(X_1) + \beta Y\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } ((X_1, X_2), Y) \ = \textbf{symm true } \alpha\ \_\ X_1 \_\ X_2\ \beta\ Y \textbf{ in } e$$
$$\textbf{let } Y \leftarrow [|\ \alpha X_1^{T?} X_2^{T?} + \beta Y\ |] \textbf{ in } e \Rightarrow$$
$$\quad \textbf{let } ((X_1, X_2), Y) \ = \ \textbf{gemm } \alpha\ \_\ (X_1, {}^{\textbf{true}}_{\textbf{false}})\ \_\ (X_2, {}^{\textbf{true}}_{\textbf{false}})\ \beta\ Y \textbf{ in } e$$

Fig. 10. Purely syntactic pattern-matching translations of matrix expressions.

with a code-formatter, the resulting code is not obviously correct and exactly what an expert would intend to write by hand, but now with the guarantees and safety of NumLin behind it. A small example is shown in Figure 14, a larger one in Figure 17.

4.2 Performance Metrics

Here, evaluate the performance of the examples from the second section. Compare with your C implementations, and perhaps as well as the straightforward math transcribed into (Matlab/R/Numpy?).

5 Discussion and Related Work

5.1 Finding Bugs in SymPy's Output

Prior to this project, we had little experience with linear algebra libraries or the problem of matrix expression compilation. As such, we based our initial NumLin implementation of a Kalman filter using BLAS and LAPACK, on a

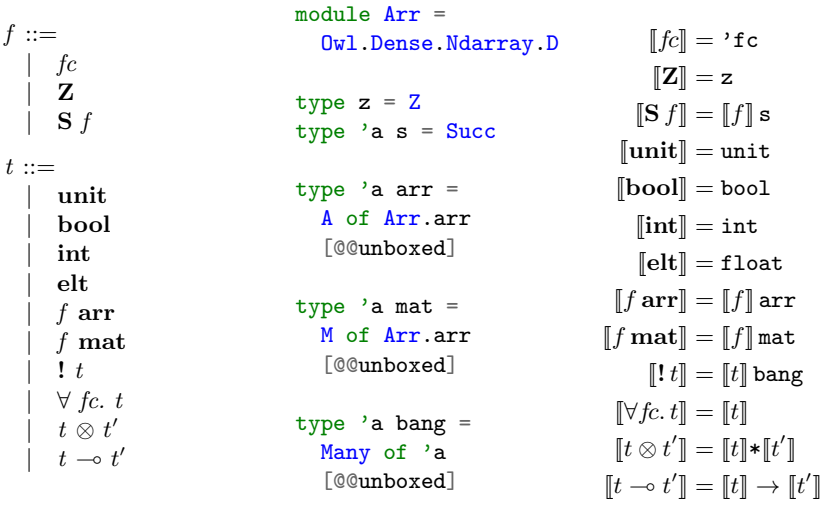

Fig. 11. NumLin's type grammar (left) and its embedding into OCaml (right).

popular GitHub gist of a Fortran implementation, one that was *automatically generated* from SymPy's matrix expression compiler [6].

Once we translated the implementation from Fortran to NumLin, we attempted to compile it and found that (to our surprise) it did not type-check. This was because the original implementation contained incorrect aliasing, unused and unnecessary temporaries, and did not adhere to Fortran's read/write permissions (with respect to `intent` annotations `in`, `out` and `inout`) all of which were now highlighted by NumLin's type system.

The original implementation used 6 temporaries, one of which was immediately spotted as never being used due to linearity. It also contained two variables which were marked as `intent(in)` but would have been written over by calls to 'gemm', spotted by the fractional-capabilities feature. Furthermore, it used a matrix *twice* in a call to 'symm', once with a read permission but once with a *write* permission. Fortran assumes that any parameter being written to is not aliased and so this call was not only incorrect, but illegal according to the standard, both aspects of which were captured by linearity and fractional-capabilities.

Lastly, it contained another unnecessary temporary, however one that was not obvious without linear types. To spot it, we first performed live-range splitting (checked by linearity) by hoisting calls to `freeM` and then annotated the freed matrices with their dimensions. After doing so and spotting two disjoint live-ranges of the same size, we replaced a call to `freeM` followed by allocating call to `copy` with one, in-place call to `copyM_to`. We believe the ability to boldly refactor code which manages memory is good evidence of the usefulness of linearity as a tool for programming.

```
  1 let lt4la_kalman ~sigma ~h ~mu ~r ~data =
  0     Examples.Kalman.it (M sigma) (M h) (M mu) (M r) (M data)
 NORMAL  test/examples_test.ml
'a mat ->
'b mat ->
'c mat ->
z mat ->
z mat -> ('a mat * ('b mat * ('c mat * (z mat * z mat)))) * (z mat * z mat)
 :merlin-type-history:
  0   let fact = Examples.Factorial.it in
 NORMAL  test/examples_test.ml
int bang -> int bang
```

Fig. 12. Using NumLin functions from OCaml.

```
  let Many x = x in
  let Many x = Many (Many x) in <exp> ⇒ <exp>

let Many x = <exp> in
let Many x = Many (Many x) in <body>
OR let Many x = Many <exp> in <body>   ⇒ let x = <exp> in <body>
OR (fun x : t -> <body>) <exp>

(* fixp = fix (f, x:t, <exp> : t') *)
let Many f = Many fixp in <body>       ⇒ let rec f x = <exp> in <body>
OR let f = fixp in <body>
```

Fig. 13. Removing redundant re-bindings during tranlation to OCaml.

5.2 Related Work

The main point we want to make is that using linear types for BLAS is an "obvious" idea, but is surprisingly under-explored.

- Rust
- ATS
- Single-assignment C
- Linear Haskell
- Bernardy and Sveningsson
- L3
- Boyland fractional permissions
- Disadvantage of graph-based approaches to matrix expression generation
- multi-stage: once dimensions known, they are fixed?

5.3 Simplicity and Further Work

We are pleasantly surprised at how simple the overall design and implementation of NumLin is, given its expressive power and usability. Indeed, the focus

```
let rec f i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    f (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row
in
f
```

Fig. 14. Recursive OCaml function for a summing over an array, generated (at *compile time*) from the code in Figure 2, passed through `ocamlformat` for presentation.

on getting a working prototype early on (so that we could test it with real BLAS/LAPACK routines as soon as possible) meant that we only added features to the type system when it was clear that they were absolutely necessary: these features were !-types and value-restriction for the `Many` constructor. Although we considered parametric polymorphism, size-types and tracking pointer identities at the type-level[5] (all of which would certainly be useful and are potential avenues for further work) it turns out these features are not necessary.

We believe we have stumbled across a particularly nice problem-solution fit: the idea of linear types for linear algebra – one that used to just be commonly believed folklore – actually has some merit and is worth exploring. We also think that this concept (and the general design of its implementation) need not be limited to linear algebra: we could conceivably 'backport' this idea to other contexts that need linearity (concurrency, single-use continuations, zero-copy buffer, streaming I/O) or combine it with dependent types to achieve even more expressive power to split up a single block of memory into multiple regions in an arbitrary manner[4].

References

1. Bierhoff, K., Beckman, N.E., Aldrich, J.: Fraction polymorphic permission inference
2. Bronstein, A., Choukroun, Y., Kimmel, R., Sela, M.: Consistent discretization and minimization of the l1 norm on manifolds. In: 3D Vision (3DV), 2016 Fourth International Conference on. pp. 435–440. IEEE (2016)
3. Kalman, R.E.: A new approach to linear filtering and prediction problems. Journal of basic Engineering **82**(1), 35–45 (1960)
4. McBride, C.: Code mesh london 2016, keynote: Spacemonads. https://www.youtube.com/watch?v=QojLQY5H0RI, accessed: 08/05/2018
5. Morrisett, G., Ahmed, A., Fluet, M.: L 3: a linear language with locations. In: International Conference on Typed Lambda Calculi and Applications. pp. 293–307. Springer (2005)
6. Rocklin, M.: Mathematically informed linear algebra codes through term rewriting. Ph.D. thesis (2013)

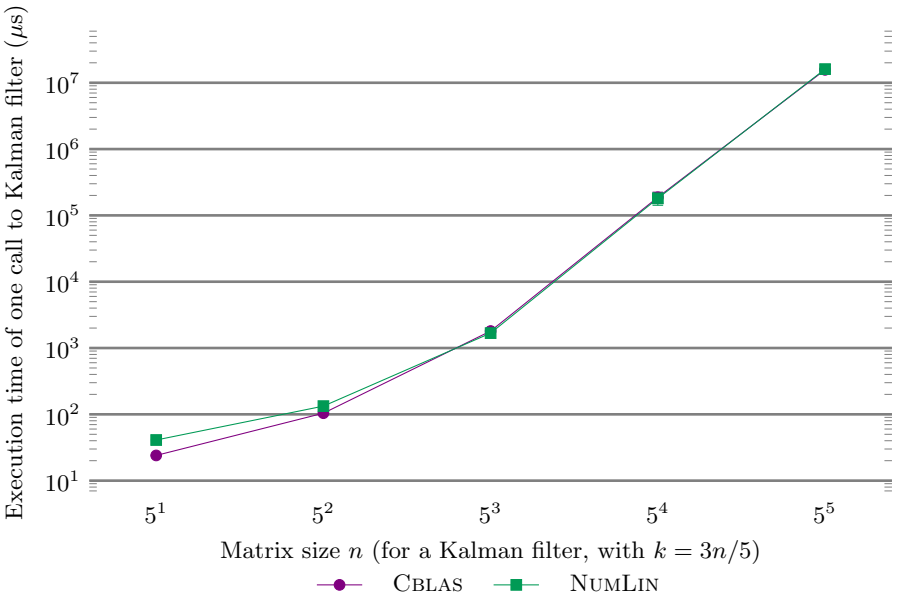

Fig. 15. Comparison of execution times (error bars are present but quite small). Small matrices and timings $n \leq 5^3$ were micro-benchmarked with the Core_bench library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.

A Appendix

A.1 Static Semantics

A.2 Dynamic Semantics

A.3 Desugaring NumLin

$$
\begin{aligned}
x[e] &\Rightarrow \textbf{get}\ \_\ x\ (e) && \text{(similarly for matrices)}\\
x[e_1] := e_2 &\Rightarrow \textbf{set}\ x\ (e_1)\ (e_2) && \text{(similarly for matrices)}
\end{aligned}
$$

$$
\begin{aligned}
pat ::= {} & () \mid x \mid\ !x \mid \textbf{Many}\ pat \mid (pat, pat)\\
\textbf{let}\ !x = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ x = e_1\ \textbf{in}\\
&\quad\ \textbf{let Many}\ x = \textbf{Many}\ (\textbf{Many}\ x)\ \textbf{in}\ e_2\\
\textbf{let Many}\langle pat_x\rangle\ = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ x = x\ \textbf{in}\\
&\quad\ \textbf{let}\ \langle pat_x\rangle\ = x\ \textbf{in}\ e_2\\
\textbf{let}\ (\langle pat_a\rangle, \langle pat_b\rangle)\ = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let}\ (a, b)\ = a\_b\ \textbf{in}\ \ \textbf{let}\ \langle pat_a\rangle\ = a\ \textbf{in}\\
&\quad\ \textbf{let}\ \langle pat_b\rangle\ = b\ \textbf{in}\ e_2\\
\textbf{fun}\ (\langle pat_x\rangle : t) \to e &\Rightarrow \textbf{fun}\ (x : t) \to \textbf{let}\ \langle pat_x\rangle = x\ \textbf{in}\ e
\end{aligned}
$$

$$
\begin{aligned}
arg ::= {} & \langle pat\rangle : t \mid\ 'x \text{ (fractional permission variable)}\\
\textbf{fun}\ \langle arg_{1..n}\rangle \to e &\Rightarrow \textbf{fun}\ \langle arg_1\rangle \to ..\ \textbf{fun}\ \langle arg_n\rangle \to e\\
\textbf{let}\ f\ \langle arg_{1..n}\rangle = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let}\ f = \textbf{fun}\ \langle arg_{1..n}\rangle \to e_1\ \textbf{in}\ e_2\\
\textbf{let}\ !f\ \langle arg_{1..n}\rangle = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ f = \textbf{Many}\ (\textbf{fun}\ \langle arg_{1..n}\rangle \to e_1)\ \textbf{in}\ e_2\\
\text{fixpoint} &\equiv \textbf{fix}\ (f, x : t, \textbf{fun}\ \langle arg_{1..n}\rangle \to e_1 : t')\\
\textbf{let rec}\ f\ (x : t)\ \langle arg_{1..n}\rangle : t' = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let}\ f = \text{fixpoint}\ \textbf{in}\ e_2\\
\textbf{let rec}\ !f\ (x : t)\ \langle arg_{1..n}\rangle : t' = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ f = \textbf{Many}\ \text{fixpoint}\ \textbf{in}\ e_2
\end{aligned}
$$

Fig. 16. Desugaring from NumLin concrete syntax to core constructs.

```
let kalman sigma h mu r_1 data_1 =
  let h, _p_k_n_p_ = Prim.size_mat h in
  let k, n = _p_k_n_p_ in
  let sigma_h = Prim.matrix k n in
  let (sigma, h), sigma_h =
    Prim.symm (Many true) (Many 1.) sigma h (Many 0.) sigma_h
  in
  let (sigma_h, h), r_2 =
    Prim.gemm (Many 1.) (sigma_h, Many false) (h, Many true) (Many 1.) r_1
  in
  let (h, mu), data_2 =
    Prim.gemm (Many 1.) (h, Many false) (mu, Many false) (Many (-1.)) data_1
  in
  let h, new_h = Prim.copy_mat_to h sigma_h in
  let r_2, new_r = Prim.copy_mat r_2 in
  let chol_r, sol_h = Prim.posv new_r new_h in
  let chol_r, sol_data = Prim.potrs chol_r data_2 in
  let () = Prim.free_mat chol_r in
  let h_sol_h = Prim.matrix n n in
  let (h, sol_h), h_sol_h =
    Prim.gemm (Many 1.) (h, Many true) (sol_h, Many false) (Many 0.) h_sol_h
  in
  let () = Prim.free_mat sol_h in
  let h_sol_data = Prim.matrix n (Many 1) in
  let (h, sol_data), h_sol_data =
    Prim.gemm (Many 1.) (h, Many true) (sol_data, Many false) (Many 0.) h_sol_data
  in
  let mu, mu_copy = Prim.copy_mat mu in
  let (sigma, h_sol_data), new_mu =
    Prim.symm (Many false) (Many 1.) sigma h_sol_data (Many 1.) mu_copy
  in
  let () = Prim.free_mat h_sol_data in
  let h_sol_h_sigma = Prim.matrix n n in
  let (sigma, h_sol_h), h_sol_h_sigma =
    Prim.symm (Many true) (Many 1.) sigma h_sol_h (Many 0.) h_sol_h_sigma
  in
  let sigma, sigma_copy = Prim.copy_mat_to sigma h_sol_h in
  let (sigma, h_sol_h_sigma), new_sigma =
    Prim.symm (Many false) (Many (-1.)) sigma h_sol_h_sigma (Many 1.) sigma_copy
  in
  let () = Prim.free_mat h_sol_h_sigma in
  ((sigma, (h, (mu, (r_2, sol_data)))), (new_mu, new_sigma)) )
in
kalman
```

Fig. 17. OCaml code for a Kalman filter, generated (at *compile time*) from the code in Figure 8, passed through `ocamlformat` for presentation.

```
static void kalman( const int n,               const int k,               const double *sigma, /* n,n */
                    const double *h, /* k,n */ const double *mu, /* n,1 */ double *r,           /* k,k */
                    double *data,    /* k,1 */ double **ret_mu,  /* k,1 */ double **ret_sigma   /* n,n */ ) {
        double* k_by_n = (double *) malloc(k * n * sizeof(double));
/*16*/  cblas_dsymm(CblasRowMajor, CblasRight, CblasUpper, k, n, 1., sigma, n, h, n, 0., k_by_n, n);
/*17*/  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, k, k, n, 1., k_by_n, n, h, n, 1., r, k);
/*18*/  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, k, 1, n, 1., h, n, mu, 1, -1., data, 1);
/*19*/  cblas_dcopy(k * n, h, 1, k_by_n, 1);
        double* k_by_k = (double *) malloc(k * k * sizeof(double));
/*20*/  cblas_dcopy(k * k, r, 1, k_by_k, 1);
/*21*/  LAPACKE_dposv(LAPACK_ROW_MAJOR, 'U', k, n, k_by_k, k, k_by_n, n);
/*23*/  LAPACKE_dpotrs(LAPACK_ROW_MAJOR, 'U', k, 1, k_by_k, k, data, 1);
        free(k_by_k);
        double* n_by_n = (double *) malloc(n * n * sizeof(double));
/*24*/  cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, n, k, 1., h, n, k_by_n, n, 0., n_by_n, n);
        free(k_by_n);
        double* n_by_1 = (double *) malloc(n * sizeof(double));
/*25*/  cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, 1, k, 1., h, n, data, 1, 0., n_by_1, 1);
        double* new_mu = (double *) malloc(n * sizeof(double));
/*26*/  cblas_dcopy(n, mu, 1, new_mu, 1);
/*27*/  cblas_dsymm(CblasRowMajor, CblasLeft, CblasUpper, n, 1, 1., sigma, n, n_by_1, 1, 1., new_mu, 1);
        free(n_by_1);
        double* n_by_n2 = (double *) malloc(n * n * sizeof(double));
/*28*/  cblas_dsymm(CblasRowMajor, CblasRight, CblasUpper, n, n, 1., sigma, n, n_by_n, n, 0., n_by_n2, n);
/*29*/  cblas_dcopy(n*n, sigma, 1, n_by_n, 1);
/*30*/  cblas_dsymm(CblasRowMajor, CblasLeft, CblasUpper, n, n, -1., sigma, n, n_by_n2, n, 1., n_by_n, n);
        free(n_by_n2);
        *ret_sigma = n_by_n;
        *ret_mu = new_mu; }
```

Fig. 18. CBLAS/LAPACKE implementation of a Kalman filter. I used C instead of Fortran because it is what Owl uses under the hood and OCaml FFI support for C is better and easier to use than that for Fortran. A distinct 'measure_kalman' function that sandwiches a call to this function with `getrusage` is omitted for brevity.